CMPTG 5 - Statistical Physics of Neural Networks

Sidharth Kannan

May 2025

1 The Neural Tangent Kernel

Over the past two weeks, we have taken a peek into the initialization behavior of neural networks. In particular, we looked at how *deep linear models* behave at initialization by computing the correlators of the preactivation distributions.

Today, we are going to expand our toolbox to incorporate the training procedure. I will mostly try and sketch out qualitative aspects of neural network training, and introduce the relevant concepts at a sufficiently high level, rather than focus on the proofs and mathematics of deep learning theory.

1.1 Linear Regression

Let's start with a classic, as usual with the intention of setting up notation and giving us a baseline to work off of. In linear regression, we are given a training set of inputs and labels, $\mathcal{D} = \{(x_i, y_i)\}$, and our goal is to find a linear model, β, such that $\beta x_i = y_i$. In general, the *features*, x_i may be vectors, in which case the model, β is also a vector. In that case, a more appropriate notation would be $\boldsymbol{\beta}^T \mathbf{x} = y_i$.

While linear regression does admit a closed form solution, it suits our purposes better to solve it through *gradient descent*. Suppose we are using the standard *mean squared error* loss.

$$\mathcal{L} = \frac{1}{2} \sum_i ||\boldsymbol{\beta}^T \mathbf{x} - y_i||_2^2 \tag{1}$$

That is, we measure the error of our model by taking the sum (or average) squared error across all data points. Then, the gradient descent algorithm says that

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \eta \nabla_{\boldsymbol{\beta}} \mathcal{L} \tag{2}$$

In the case of the MSE loss, we can explicitly evaluate this gradient. Applying the chain rule, we have that

$$\begin{aligned}\nabla_{\boldsymbol{\beta}} \mathcal{L} &= \sum_i (y_i - \boldsymbol{\beta}^T \mathbf{x}) \cdot \nabla_{\boldsymbol{\beta}} (\boldsymbol{\beta}^T \mathbf{x}) \\ &= \sum_i (y_i - \boldsymbol{\beta}^T \mathbf{x}) \cdot \mathbf{x}\end{aligned}$$

Okay, so we can do this. The trouble is that our model is linear, and so if our data is non-linear, there is simply no set of parameters that can fit it. We saw this idea earlier in the case of deep linear networks, in that the network loses some expressive power when you remove the non-linearities. There, we were okay with it because we could still study the behavior of the network and glean meaningful insights that would apply to non-linear networks, but from a practitioner's point of view, the linearity limitation is a major buzzkill.

1.2 Kernel Methods

One way of handling this is with a *kernel method*. The idea is this: For some non-linear regression or classification problem, can we map the data somehow into a higher dimension space where things become linear?

Let's say we want to train a classifier to distinguish whether or not somebody is a homeowner, and our two input features are age and income. We might expect that older people are more likely to own a home, as are wealthier people. We might further expect that older people have greater incomes, and so our dataset may look something like the toy model in Fig. 1.

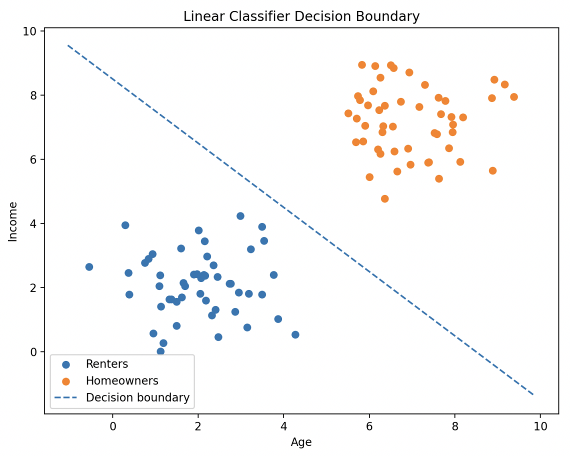


Figure 1: An example of a simple, linearly separable problem

In this example, the dataset is very easy to separate, and so a simple, linear classifier suffices. We can use the line $y = 10 - x$ as our decision boundary, and classify our data points by simply testing, is $y > x - 10$? If so, our point is very likely a homeowner, and if not it's very likely that they are not.

Of course, this is a highly simplified model, and real world datasets do not often look like this. We might encounter some other dataset like the dataset in Fig. 2, which is clearly not linearly separable. No matter where we choose our dividing line, we will not get a clean separation.

Looking at this dataset, it should be clear that we could get away with an only marginally more clever classifier: one that classifies the data point based on the radius that we are at, not the x or y coordinate directly.

If we project into the 3D space, $(x, y) \to (x, y, \sqrt{x^2 + y^2})$, we see (Fig. 3) that the data is now easily separable by a linear model.

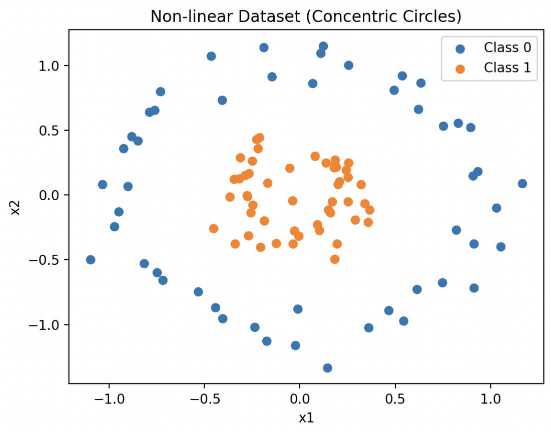


Figure 2: A non-linear dataset.

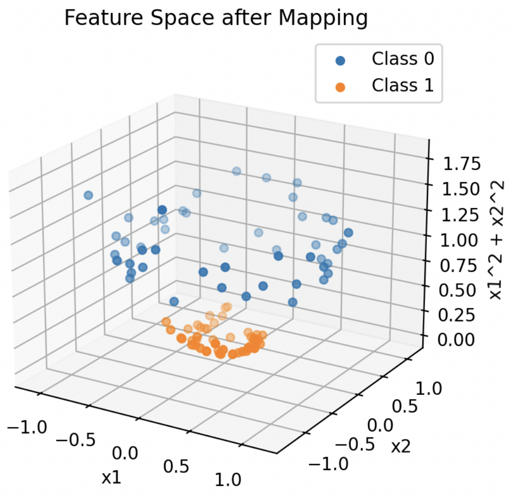


Figure 3: The set is easily linearly separable now.

The general name for this type of transformation is a *kernel*, denoted ϕ. Notice how the structure of our model changes slightly. Instead, of a linear model, we get a model that is *linear* in the weights, but *non-linear* in the output. To see that, see how our loss function changes:

$$\mathcal{L} = \frac{1}{2} \sum_i ||\boldsymbol{\beta}^T \phi(\mathbf{x}) - y_i||_2^2 \tag{3}$$

There are two problems with applying this method more generally. The first is that we had to hand-craft this transformation. The second is that in order to compute this transformation, we had to increase the dimensionality of our data. Not a problem in our 3 dimensional setting, but you could imagine that on a more realistic dataset, where we might start in higher dimensions, and then have to increase the dimensionality a lot more than we did in this example, we might run into problems.

To see this more clearly, we can take the example of the following transformation:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \to \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix} \tag{4}$$

This is a polynomial transformation of the input, and we see that even though our input was 2D, we suddenly have a 5-dimensional feature vector. In general, polynomial transformations like this scale very poorly. Say our input was an MNIST image, with 576 elements per input. If we wanted all degree 2 combinations of this, we would get 576^2 elements in our feature, which is rather large. In general, for a d-dimensional input, a degree k polynomial transform will result in a d^k scaling law. This is one example of the curse of dimensionality.

1.3 The Kernel Trick

It turns out that this isn't as big of a problem as it seems. In many cases, we don't truly care about the actual high dimensional vectors, $\phi(x), \phi(x')$, but instead we just want to compute their inner product, $\phi(x)^T\phi(x')$. The kernel trick gives us a way to do this without ever actually computing the high dimensional vectors.

So, we define the *kernel matrix*

$$K_{ij} \equiv K(x_i, x_j) = \phi(x_i)^T \phi(x_j) \tag{5}$$

The kernel matrix is obviously symmetric, by virtue of they symmetry of the dot product. The kernel can also be shown to be positive semi-definite.

The key point here is that for certain choices of kernel function ϕ, we can compute this inner product without actually computing $\phi(x)$. Let's see an example. Consider the degree two polynomial kernel:

Let $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \in \mathbb{R}^2$. Consider the second-degree polynomial feature map

$$\phi : \mathbb{R}^2 \longrightarrow \mathbb{R}^3, \qquad \phi\big([x_1, x_2]\big) = \begin{bmatrix} x_1^2 \\ \sqrt{2}\, x_1 x_2 \\ x_2^2 \end{bmatrix}.$$

Then, if we map x and z to this higher dimensional feature space, and compute their inner product, we get

$$\phi(x) \cdot \phi(z) = x_1^2 z_1^2 + 2\, x_1 x_2\, z_1 z_2 + x_2^2 z_2^2.$$

We can, however, instead define

$$K(x, z) \;=\; \big(x^\top z + c\big)^2.$$

With $c = 0$, this reduces to

$$K(x,z) = (x_1 z_1 + x_2 z_2)^2 = x_1^2 z_1^2 + 2\, x_1 x_2\, z_1 z_2 + x_2^2 z_2^2 = \phi(x) \cdot \phi(z).$$

Thus, instead of computing $\phi(x)$ and $\phi(z)$ in $\mathbb{R}^3$ and taking their dot product, we directly evaluate

$$K(x,z) = (x^\top z)^2,$$

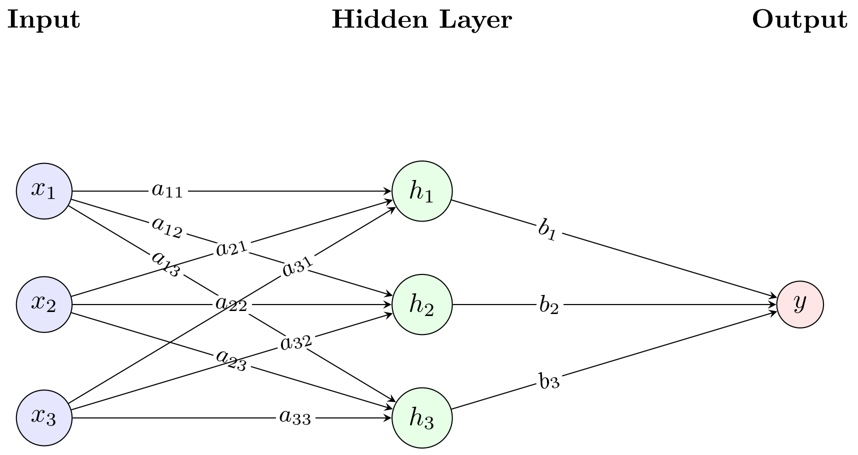
in $\mathcal{O}(d)$ time (here $d = 2$), without ever forming the 3-dimensional vectors. This general idea is known as the *kernel trick*. While it doesn't produce much savings in this particular case, you can see how it would, say for higher dimensional polynomial feature maps, allow much faster computations.

In general, for a d-degree polynomial, the kernel trick becomes $(x^T z + c)^d$, i.e. constant time. How can we use this kernel machine to do useful tasks? Well, if you recall, the kernel is supposed to measure similarity, so we do inference in the following way:

$$y(x) = \sum_i K(x_i, x) y_i$$

1.4 Neural Networks

Now, let's consider a neural network, and see how we can apply this idea to it. In particular, consider a simple, 2-layer neural network. As per usual, we make these assumptions to simplify the analysis. If you are curious, the results we will discuss today do hold up for deep feed-forward networks, and for other neural network architectures.



Let's say the input is $x \in \mathbb{R}^d$, and we have m neurons in the hidden layer. Then, we can write the neural network function as

$$f_\theta(x) = \frac{1}{\sqrt{m}} \sum_{i=1}^{m} b_i \sigma \left(a_i^T x \right)$$

Note that here, a_i is vector, since it is the i-th row on the matrix representing the first layer. Let's say that again, we are training this network with gradient descent on the mean squared error, so again the loss is

$$\mathcal{L} = \frac{1}{N} \sum_i ||f_\theta(x_i) - y_i||_2^2 \tag{6}$$

As before, we can write the gradient descent update.

$$\theta_{t+1} = \theta_t - \eta \sum_i (f_\theta(x_i) - y_i)\nabla_\theta f_\theta(x_i) \tag{7}$$

Here, we just used the chain rule to express the gradient of the loss explicitly. Note that if we look at the terms in the summation, the first one, $f_\theta(x_i) - y_i$, is the same as during linear regression, but now, since our model is *not* linear in θ, the second factor, $\nabla_\theta f_\theta(x_i)$, is not just x_i. In particular, it is a function of θ, and so it changes over the course of training.

Now, in principle, this could be the end for us. It could be the case that the way that this changes during training is so complicated that it borders on intractable. However, there is an observation that we can make that saves the day. Recall how in the last few weeks, we used the assumption of infinite width to our advantage. We are going to pull that trick again.

Say that we have a very, very wide neural network, and as per usual, we sample its parameters from a Gaussian distribution. Question: As I train this network under gradient descent, how do I expect the entries of the matrix to change? It turns out that as the network gets wider, the matrix entries change less and less. This is a statement that is a) not very precise, and b) not universally true, but the idea is that as the network gets wider, the loss landscape get more convex, and the contributions of the individual matrix entries become less important to the output, so they change very little. In the literature, this is known as the "lazy" regime, or the "kernel" regime. The first name is fairly self explanatory. The second will become clearer in a moment. But okay, let's say that we are working in this regime, where the initialization parameters, θ_0 are very close to the trained network parameters, θ. Then, I can employ a first order Taylor expansion:

$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)^T\Big|_{\theta=\theta_0}(\theta - \theta_0) \tag{8}$$

Is this linear? Yes, it is linear in θ, but no, it is not linear in x. Does that remind you of something? Yes, it is the same pattern we saw in our kernel machine. We make the identification that $\phi(x) = \nabla_\theta f_{\theta_0}(x)$, and give it the fancy name, the *neural tangent kernel.* Notice that this derivative is evaluated at θ_0, so it does not change over the course of training.

Okay, so what actually is this gradient. Recall that the whole premise of kernel machines was that we could compute this in closed form. So, let's compute the gradients for our two layer network setup before. There, we had two sets of parameters, the a_i's and the b_i's

$$\begin{aligned}\nabla_{a_i} f_\theta(x) &= \nabla_{a_i}\left[\frac{1}{\sqrt{m}}\sum_{i=1}^{m} b_i \sigma\left(a_i^T x\right)\right] \\ &= \frac{1}{\sqrt{m}} b_i \cdot \sigma'\left(a_i^T x\right) x\end{aligned}$$

Similarly for the b_i's,

$$\begin{aligned}\nabla_{b_i} f_\theta(x) &= \nabla_{b_i}\left[\frac{1}{\sqrt{m}}\sum_{i=1}^{m} b_i \sigma\left(a_i^T x\right)\right] \\ &= \frac{1}{\sqrt{m}}\cdot\sigma\left(a_i^T x\right)\end{aligned}$$

Then, our kernel matrix entry is just the inner product of the sum of these, evaluated at two different x's. Here, I use $\langle x, x'\rangle$ to denote the dot product.

$$K^{(a)}(x,x') = \frac{1}{m}\sum_{i=1}^{m} b_i^2 \sigma'(a_i^T x)\sigma'(a_i^T x')\langle x, x'\rangle \tag{9}$$

$$K^{(b)}(x,x') = \frac{1}{m}\sum_{i=1}^{m} \sigma(a_i^T x)\sigma(a_i^T x') \tag{10}$$

Now, the a_i's and b_i's are random variables, but if we let $m \to \infty$, we can apply the law of large numbers to say that the average of the sum of a bunch of independent RVs converges almost surely to their expectation. That is,

$$K^{(a)}(x,x') = \mathbb{E}\left[b^2\sigma'(a_i^T x)'\sigma(a_i^T x')\langle x, x'\rangle\right] \tag{11}$$

$$K^{(b)}(x,x') = \mathbb{E}\left[\sigma(a_i^T x)\sigma(a_i^T x')\right] \tag{12}$$

Now, it turns out these expectations can be done in closed form for a wide range of activation functions. For example, if the activation function is a ReLU,

$$K^{(a)}(x,x') = \frac{\langle x, x'\rangle \mathbb{E}[b^2]}{2\pi}(\pi - \theta(x,x')) \tag{13}$$

where $\theta(x,x')$ denotes the angle between the vectors x, x'. Remember, you can compute that angle using the dot product. Similarly,

$$K^{(b)}(x,x') = \frac{||x||\cdot||x'||\mathbb{E}[a^2]}{2\pi d}((\pi - \theta(x,x'))\cos\theta + \sin\theta) \tag{14}$$

The actual form of the expectation isn't very important. The point is that it can be computed efficiently, as long as we know can compute dot products of the data points. Okay, so what is this all good for? First off, NTKs are far better in practice than most traditional kernel methods. We can understand this, since the kernel they are representing does not correspond to some hand crafted feature transform, but instead to an infinite width neural network. NTKs don't, however, generally outperform actual deep neural networks. The reason for this is because, as we saw in the last class, finite width fluctuations are a key part of the expressive power of NNs. The infinite width approximation turns out to be too strong.

The next reason we care about these NTKs is that they are a supremely useful tool for understanding the *dynamics* of neural networks. In the previous lectures, we did a lot of study on neural network statics, but we know that training is a vital component of NN performance, so we need to develop tools to understand how training affects networks. I will show you a simple example of how the NTK enables that now.

Let's imagine a neural network being trained under gradient descent.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L} \tag{15}$$

We can rearrange this to give

$$\frac{\theta_{t+1} - \theta_t}{\eta} = -\nabla_\theta \mathcal{L} \tag{16}$$

and if we take the limit where η is small, we get the following differential equation

$$\frac{d\theta}{dt} = -\nabla_\theta \mathcal{L} \tag{17}$$

This is called *gradient flow* dynamics. Again, let's look at the MSE loss.

$$\mathcal{L}_\theta = \frac{1}{2}||f_\theta(x) - y||^2 \tag{18}$$

where $f_\theta(x)$ is my network prediction when the parameters are θ. Again, if we compute the gradient now, we get

$$= (f_\theta(x) - y)\nabla_\theta f_\theta(x) \tag{19}$$

Plugging that in to the differential equation, we get

$$\frac{d\theta}{dt} = -(f_\theta(x) - y)\nabla_\theta f_\theta(x) \tag{20}$$

This shows the dynamics in parameter space. That is, it shows us how the weights change as a function of time. We can also look at the output dynamics. That is, we can study

$$\begin{aligned} \frac{df_\theta}{dt} &= \left[\nabla_\theta f_\theta(x)\right]^T \frac{d\theta}{dt} \\ &= -\left[\nabla_\theta f_\theta(x)^T \cdot \nabla_\theta f_\theta(x)\right](f_\theta(x) - y) \end{aligned}$$

But look at this! The term in the square brackets is just the product between two different matrices, $\nabla_\theta f_\theta(x)$, evaluated on different samples! This is just the NTK! It's true that we are evaluating this at θ_t, but we can approximate it with θ_0, since we argued that the parameters don't change very much. This is a point worth emphasizing. In the infinite width limit, it is exactly true that the NTK is constant, and that it is deterministic. The determinism comes from the law of large numbers argument we made earlier, and the point is that in the infinite width limit, it doesn't matter that the network parameters are randomly sampled; the NTK converges to the expectation. The constancy comes from the fact that this Taylor approximation that we made is only good in the large width limit; otherwise the network parameters change appreciably and our argument breaks down.

So, notationally, we can write this as

$$\frac{df_\theta}{dt} = -\Theta(f_\theta(x) - y) \rightarrow \frac{d}{dt}(f_\theta - y) = -\Theta(f_\theta(x) - y) \tag{21}$$

where we take that step, since y is a constant. Then, we easily recognize that this is solved by an exponential.

$$f_\theta(t) - y = (f_\theta(0) - y)e^{-Kt} \tag{22}$$

Okay, so our training error follows this exponential decay. In particular, in the case where the network is highly overparametrized, the kernel becomes positive definite, and so the eigenvalues become strictly positive. Taking the eigendecomposition of the kernel matrix, we get

$$f_\theta(t) - y = (f_\theta(0) - y)e^{-Kt} = (f_\theta(0) - y)e^{-t\sum \lambda_i v_i v_i^T} = (f_\theta(0) - y)\prod_{i=1}^{n} e^{-\lambda_i v_i v_i^T} \tag{23}$$

where λ_i's are the eigenvalues of the NTK, and v_i's are the eigenvectors. And so, it becomes clear that the spectrum of the NTK tells us something very important about how the training error of our network converges. The eigenvalues of the NTK tell us the rate of convergence along different directions in the data space, and the smallest eigenvalue and its corresponding eigenvector tell us the direction in which the training error will converge most slowly.

As a second to last little note, the name *neural tangent kernel* is worth breaking down a little. It is *neural* because it is a kernel that corresponds to a neural network. It is *tangent*, since the kernel function is the gradient with respect to the parameters of the network output, and it is *kernel* for reasons that should be clear by now. However, we don't really care about it because it makes a good kernel machine. Kernel methods have largely fallen out of fashion since deep learning became a thing. It is this behavior of the NTK, that it governs the time evolution of network outputs (and more generally any network observable) that makes it interesting.

To those of you who know about Hamiltonians, the NTK is perhaps better viewed as a sort of Hamiltonian, that governs the time evolution of the network, and this is the property that truly makes it a powerful tool for the study of neural networks.

Finally, I will conclude by stating that NTKs are not limited to two layer feed forward networks. Jacot et. al 2018 show the general way to compute the NTK for an arbitrary depth neural network, and soon after, corresponding kernels were derived for CNNs, GNNs, and other network architectures.